Pràctica OpenMP

FeedForward Neural Network

Computació d'Altes Prestacions. MatCAD

Natan Sisoev (1706198) i Ferran Villarta (1704051)

19 d'octubre de 2025

Sumari

1 Introducció i Context General

1.1 Motivació i Objectius

L'entrenament de xarxes neuronals és una tasca computacionalment intensiva que requereix múltiples iteracions sobre grans conjunts de dades. En el context de la computació d'altes prestacions, la paral·lelització d'aquests algoritmes és essencial per reduir els temps d'execució i permetre l'entrenament de models més complexos.

Els objectius principals d'aquesta pràctica són:

1. Identificar les seccions del codi d'entrenament susceptibles de ser paral·lelitzades
2. Mesurar el speedup i l'eficiència de cada paral·lelització individualment
3. Determinar la combinació òptima de paral·lelitzacions
4. Analitzar l'escalabilitat del codi optimitzat sota diferents configuracions
5. Estudiar l'impacte de paràmetres com el nombre d'epochs, neurones i threads

1.2 Conceptes Fonamentals

1.2.1 Speedup

El speedup (S) mesura la millora de rendiment obtinguda amb la paral·lelització:

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}} \tag{1}$$

on T_{seq} és el temps d'execució seqüencial i T_{par} és el temps amb p processadors.

1.2.2 Eficiència

L'eficiència (E) normalitza el speedup pel nombre de processadors utilitzats:

$$E = \frac{S}{p} = \frac{T_{\text{seq}}}{p \cdot T_{\text{par}}} \tag{2}$$

Un valor d'eficiència proper a 1 indica un ús òptim dels recursos.

1.2.3 Llei d'Amdahl

La Llei d'Amdahl estableix el límit teòric del speedup en funció de la fracció paral·lelitzable del codi (f):

$$S_{\max} = \frac{1}{(1-f) + \frac{f}{p}} \tag{3}$$

Aquesta llei prediu que la part seqüencial del codi limitarà sempre el speedup màxim assolible.

1.2.4 Llei de Gustafson

La Llei de Gustafson ofereix una perspectiva més optimista, considerant que la mida del problema pot créixer amb els recursos disponibles:

$$S(p) = p - \alpha(p-1) \tag{4}$$

on α és la fracció seqüencial del temps total d'execució.

1.3 Eines i Tecnologies

1.3.1 OpenMP

OpenMP (Open Multi-Processing) és una API que suporta programació paral·lela multi-thread en memòria compartida. Utilitza directives de compilador (`#pragma`) per indicar regions de codi paral·lelitzables.

1.3.2 SLURM

SLURM (Simple Linux Utility for Resource Management) és un sistema de gestió de recursos i planificació de tasques utilitzat en clusters de càlcul d'altes prestacions. Permet:

- Assignar recursos computacionals exclusius
- Gestionar cues de treballs

- Monitoritzar l'execució de tasques
- Distribuir càrregues de treball de manera eficient

1.4 Arquitectura del Projecte

El projecte està organitzat en diversos directoris i fitxers segons la seva funcionalitat:

- **main.c** i **main.h**: Programa principal que coordina l'execució del projecte.
- **training**/: Implementació de l'algorisme de backpropagation (`training.c`, `training.h`).
- **layer**/: Gestió de les capes de la xarxa neuronal (`layer.c`, `layer.h`).
- **initialize**/: Funcions d'inicialització de la xarxa (`initialize.c`, `initialize.h`).
- **randomizer**/: Funcions per generar dades aleatòries (`randomizer.c`, `randomizer.h`).
- **common**/: Funcions i definicions comunes (`common.c`, `common.h`).
- **configuration**/: Fitxers de configuració i definicions (`config.c`, `config.h`, `configfile.txt`).
- **datasets**/: Conjunts de dades d'entrada, com `optdigits.tra` i `optdigits.cv`.
- **TESTS**/: Conté diferents tests del projecte:
  - **TEST_001**, **TEST_002**, **TEST_003**: Cada test inclou fitxers `run.py`, `results.md` i sortides en subdirectoris `OUT/A-E`.
  - Alguns tests contenen configuracions específiques dins `configuration/`.
- Altres fitxers rellevants: `README.md`, `README.txt`, fitxers de treball com `seq.sub`, `scheduler.sub` i fitxers de sortida `slurm-*.out`.

Aquesta estructura facilita la separació clara de funcions i components, permetent una organització modular del codi i la possibilitat de fer tests independents de cada part del projecte.

2 Metodologia Experimental

2.1 Entorn d'Execució

Els experiments s'han realitzat en un cluster de computació amb les següents característiques:

Component	**Especificació**
Sistema Operatiu	Linux
Compilador	GCC 13.2.1
Scheduler	SLURM
Partició 1	`nodo.q` (max 12 threads)
Partició 2	`new-nodo.q` (max 64 threads)
Flags de compilació	`-Ofast -fopenmp`

Taula 1: Configuració de l'entorn d'execució

2.2 Mesura de Temps

Per mesurar amb precisió el temps d'execució de cada secció paral·lelitzada, s'utilitza la funció `omp_get_wtime()` d'OpenMP, que proporciona temps de paret amb alta resolució:

```
1 double start_time = omp_get_wtime();
2
3 #pragma omp parallel for
4 for (int j = 0; j < num_neurons[0]; j++)
5     lay[0].actv[j] = input[i][j];
6 printf("FI: %18.16f\n", omp_get_wtime() - start_time);
```

Listing 1: Mesura de temps amb OpenMP

Aquesta tècnica permet:

- Aïllar el temps de cada paral·lelització
- Calcular speedups individuals amb precisió

- Identificar colls d'ampolla
- Comparar versions seqüencials i paral·leles

2.3 Automatització amb Python

S'han desenvolupat scripts en Python per automatitzar completament el procés experimental:

2.3.1 Script `run.py`

Aquest script genera i executa automàticament tots els jobs necessaris:

```
# Iteracio sobre totes les combinacions
for partition in partitions:
    for threads in thread_counts:
        for epochs in epoch_counts:
            for neurons in neuron_counts:
                # Modificar scheduler.sub
                # Modificar configfile.txt
                # Enviar job amb sbatch
                # Esperar finalizacio amb squeue
```

Listing 2: Exemple sintètic de generació automàtica de jobs en el TEST_003

2.3.2 Script `run_test.py`

L'script `run_test.py` actua com a punt d'entrada general per a tots els tests. La seva funció principal és cridar l'script `run.py` corresponent a cada test, transmetent-li els arguments necessaris (subcarpeta, mode i variacions):

```
# Test 001: Totes les combinacions de flags
python3 run_test.py 001 A ea

# Test 002: Speedup individual
```

```
python3 run_test.py 002 B ea

# Test 003: Configuracions variables
python3 run_test.py 003 C ea PTEN
```

Listing 3: Execució dels tests

Els paràmetres disponibles són:

- **SUBFOLDER**: Subcarpeta per organitzar els resultats
- **MODE**: `e` (executar), `a` (analitzar), o `ea` (ambdós)
- **VARIATE**: `P` (particions), `T` (threads), `E` (epochs), `N` (neurones)

Flux de coordinació entre scripts i SLURM:

1. **run_test.py**:
   - Rep el número de test i els paràmetres.
   - Localitza i crida el `run.py` corresponent amb els arguments indicats.
   - Actua com a wrapper general per executar qualsevol test de manera uniforme.
2. **run.py** dins de cada test:
   - Genera totes les combinacions possibles de flags, threads, epochs i neurones.
   - Copia i modifica els fitxers de configuració per cada combinació.
   - Modifica temporalment el `scheduler.sub` per ajustar nombre de threads i partició.
   - Submet cada combinació com un job a SLURM amb `sbatch scheduler.sub ...`.
   - Fa *polling* amb `squeue` per esperar que tots els jobs acabin.
   - Analitza els fitxers `.out` generats:

- Extrau metadades: número de threads, epochs, neurons, partició i flags.
- Llegeix temps d'execució per secció paral·lelitzada.
- Calcula *speedup*, eficiència i altres mètriques agregades.
- Desa els resultats en `results.md`.

3. **scheduler.sub** (script SLURM):

   - Cada job rebut per SLURM és autònom.
   - Compila el codi amb les flags indicades i OpenMP (`-fopenmp`).
   - Redirecciona tota la sortida a un fitxer `.out` específic.
   - Escriu metadades al fitxer `.out` (threads, epochs, partició, job id, flags, servidor).
   - Executa l'executable tantes vegades com indica `REPEAT`, marcant cada execució amb `#START` i `#END`.

Flux de dades resumit:

Script	Funció
run_test.py	Wrapper general: selecciona test i passa arguments
run.py	Genera configuracions, envia jobs a SLURM, espera i analitza resultats
scheduler.sub	Job SLURM: compila, executa, escriu metadades i dades en .out
`.out`	Conté temps d'execució, metadades i resultats parcials
results.md	Conté tots els resultats agregats amb speedup i eficiència

Comentaris addicionals:

- La variable `VARIATE` permet automatitzar proves amb múltiples configuracions sense intervenció manual.
- Les particions del servidor (`nodo.q`, `new-nodo.q`, etc.) indiquen quins nodes SLURM utilitza per executar els jobs, permetent comparar rendiment entre servidors o cues diferents.

- L'ús d'OpenMP i la paral·lelització s'activa segons les flags definides per cada test.
- El procés complet permet reproduir experiments amb centenars de combinacions de manera fiable i organitzada.

2.4 Format dels Fitxers de Sortida

Cada execució genera un fitxer `.out` amb metadades i resultats:

```
---
test_number: 002
date: 2025-10-17 01:04:34
job_id: 123456
output_directory: TESTS/TEST_002/OUT/A
compilation_flags: ALL
num_threads: 12
num_epochs: 10
num_neurons: 135
server: Wilma
partition_file: nodo.q
---
#START:1
FI: 0.101662
FPL: 1.124091
BPE: 0.092299
BPO: 0.902893
BPH: 0.359971
UWW: 0.335285
UWB: 0.185019
12  3.456789
#END:1
```

Listing 4: Exemple de fitxer de sortida

2.5 Variabilitat del Servidor

Un dels reptes principals de l'experimentació en entorns compartits és la variabilitat en els temps d'execució. S'han observat:

- Desviacions estàndard de fins a 2.8 segons en execucions idèntiques
- Desviacions relatives del 40% en alguns casos
- Ràtios de rang (max/min) de fins a 9.16x

Per mitigar aquest efecte:

- S'han realitzat múltiples repeticions (fins a 100 en alguns tests)
- S'han calculat mitjanes i desviacions estàndard
- S'ha considerat que millores inferiors al 10% no són estadísticament significatives

2.6 Anàlisi de les Paral·lelitzacions

2.6.1 Feed Input (FI)

```
1 void feed_input(int i) {
2     #pragma omp parallel for
3     for (int j = 0; j < num_neurons[0]; j++)
4         lay[0].actv[j] = input[i][j];
5 }
```

Listing 5: Example Code

Intuïció: Cada iteració `j` copia un valor d'entrada a una neurona diferent. Com que cada thread escriu en `lay[0].actv[j]` amb un `j` diferent, no hi ha solapament d'escriptures.

Condició de cursa: NO. Cada thread accedeix a posicions de memòria diferents.

2.6.2 Forward Propagation Layers (FPL)

```
void forward_prop() {
    for (int i = 1; i < num_layers; i++) {
        #pragma omp parallel for
        for (int j = 0; j < num_neurons[i]; j++) {
            lay[i].z[j] = lay[i].bias[j];
            for (int k = 0; k < num_neurons[i - 1]; k++)
                lay[i].z[j] += ((lay[i - 1].out_weights[j *
                    num_neurons[i - 1] + k])
                                * (lay[i - 1].actv[k]));

            if (i < num_layers - 1)
                lay[i].actv[j] = ((lay[i].z[j]) < 0) ? 0 :
                    lay[i].z[j];
            else
                lay[i].actv[j] = 1 / (1 + exp(-lay[i].z[j]));
        }
    }
}
```

Listing 6: Example Code

Intuïció: Dins d'una mateixa capa, cada neurona `j` calcula la seva activació de forma independent. Només necessita llegir les activacions de la capa anterior (ja calculades) i els pesos.

Funcionament:

- Cada thread calcula una neurona diferent (`j` diferent)
- Escriu només a `lay[i].z[j]` i `lay[i].actv[j]` (posicions úniques)

- Llegeix `lay[i-1].actv[k]` i `lay[i-1].out_weights[...]` (lectures compartides, però segures)

Condició de cursa: NO. Cada thread escriu en posicions diferents de memòria.

2.6.3 Backpropagation Errors (BPE)

```
#pragma omp parallel for
for (int j = 0; j < num_neurons[num_layers - 1]; j++) {
    lay[num_layers - 1].dz[j] =
        (lay[num_layers - 1].actv[j] - desired_outputs[p][j]) *
        (lay[num_layers - 1].actv[j]) * (1 - lay[num_layers -
            1].actv[j]);
    lay[num_layers - 1].dbias[j] = lay[num_layers - 1].dz[j];
}
```

Listing 7: Example Code

Intuïció: Cada neurona de sortida calcula el seu error local de forma independent.

Condició de cursa: NO. Cada `j` és diferent.

2.6.4 Backpropagation Output Layer (BPO)

```
for (int j = 0; j < num_neurons[num_layers - 1]; j++) {
    #pragma omp parallel for
    for (int k = 0; k < num_neurons[num_layers - 2]; k++) {
        lay[num_layers - 2].dw[j * num_neurons[num_layers - 2] +
            k] =
            (lay[num_layers - 1].dz[j] * lay[num_layers -
                2].actv[k]);
        lay[num_layers - 2].dactv[k] =
            lay[num_layers - 2].out_weights[j *
                num_neurons[num_layers - 2] + k]
```

```
            * lay[num_layers - 1].dz[j];
    }
}
```

Listing 8: Example Code

Intuïció: Cada `k` calcula el gradient d'un pes diferent.

Problema del codi: La línia `lay[num_layers - 2].dactv[k] = ...` hauria de ser `+=` perquè múltiples `j` contribueixen al mateix `k`. Però com la paral·lelització és sobre `k` (dins d'un `j` fix), no peta, tot i que el resultat és incorrecte si hi ha múltiples neurones de sortida.

Condició de cursa: NO dins del bucle paral·lelitzat, però el codi és conceptualment incorrecte.

2.6.5 Backpropagation Hidden Layers (BPH)

```
for (int i = num_layers - 2; i > 0; i--) {
    #pragma omp parallel for
    for (int j = 0; j < num_neurons[i]; j++) {
        lay[i].dz[j] = (lay[i].z[j] >= 0) ? lay[i].dactv[j] : 0;

        for (int k = 0; k < num_neurons[i - 1]; k++) {
            lay[i - 1].dw[j * num_neurons[i - 1] + k] =
                lay[i].dz[j] * lay[i - 1].actv[k];
            if (i > 1) {
                #pragma omp critical
                lay[i - 1].dactv[k] +=
                    lay[i - 1].out_weights[j * num_neurons[i - 1]
                        + k]
                    * lay[i].dz[j];
            }
        }
```

```
16
17          lay[i].dbias[j] = lay[i].dz[j];
18      }
19  }
```

Listing 9: Example Code

Intuïció: Cada neurona `j` processa el seu error de forma independent. Escriu en posicions úniques excepte quan propaga l'error a la capa anterior.

Funcionament:

- Cada thread (diferent `j`) calcula:
  - `lay[i].dz[j]`: Error local (únic per thread)
  - `lay[i-1].dw[j * ... + k]`: Gradients dels pesos (regió única: cada `j` té la seva franja de `dw`)
  - `lay[i].dbias[j]`: Gradient del bias (únic per thread)
- Però múltiples `j` volen sumar a `lay[i-1].dactv[k]`

Per què cal `#pragma omp critical`:

- La línia `lay[i-1].dactv[k] += ...` és problemàtica
- Múltiples threads (diferents `j`) modifiquen el mateix `dactv[k]`
- Exemple: Thread 0 (j=0) i Thread 1 (j=1) volen fer `dactv[3] += ...`
- L'operació `+=` no és atòmica: llegir → sumar → escriure
- Sense sincronització, els threads poden sobreescriure's mútuament

Per què `if (i > 1)`: Si `i = 1`, la capa anterior és la input layer (`i-1 = 0`), que és fixa i no cal calcular el seu error.

Speedup excel·lent (9.55): Temps seqüencial: 3.156s (30.5% del total). La regió crítica afecta poc perquè:

- Només s'executa quan `i > 1`
- El bucle sobre `k` fa molt treball abans/després de la regió crítica

2.6.6 Update Weights (UWW)

```
for (int i = 0; i < num_layers - 1; i++) {
    #pragma omp parallel for
    for (int j = 0; j < num_neurons[i + 1]; j++)
        for (int k = 0; k < num_neurons[i]; k++)
            lay[i].out_weights[j * num_neurons[i] + k] =
                lay[i].out_weights[j * num_neurons[i] + k]
                - alpha * lay[i].dw[j * num_neurons[i] + k];
}
```

Listing 10: Example Code

Intuïció: Cada neurona `j` actualitza els seus propis pesos. Com que cada `j` té la seva franja de `out_weights[j * ... + k]`, no hi ha solapament.

Condició de cursa: NO. Cada thread escriu en una regió diferent de `out_weights`.

2.6.7 Update Biases (UWB)

```
#pragma omp parallel for
for (int j = 0; j < num_neurons[i]; j++)
    lay[i].bias[j] = lay[i].bias[j] - (alpha * lay[i].dbias[j]);
```

Listing 11: Example Code

Intuïció: Cada neurona actualitza el seu propi bias.

Condició de cursa: NO. Cada `j` és diferent.

3 Resultats Experimentals

3.1 TEST_001: Anàlisi de Combinacions

3.1.1 Objectiu

Determinar quina combinació de paral·lelitzacions és la més eficient mitjançant força bruta, provant les $2^7 = 128$ combinacions possibles dels 7 tags disponibles.

3.1.2 Metodologia

- Utilitzant la llibreria `itertools` de Python, es generen totes les combinacions possibles
- Cada combinació es compila amb els seus flags corresponents
- S'executa el programa i es mesura el temps total d'execució
- Es calcula la mitjana, desviació estàndard, mínim i màxim per cada combinació

3.1.3 Execució

El test s'executa amb:

```
python3 run_test.py 001 [SUBFOLDER] [MODE]
```

Listing 12: Example Code

On:

- `SUBFOLDER`: Nom del subfolder dins de `OUT/` (per exemple, `A`, `B`, etc.)
- `MODE`: `e` (executar), `a` (analitzar) o `ea` (ambdós)

3.1.4 Resultats del Test A (100 repeticions)

El test més gran (Test A) va executar cada combinació 100 vegades durant gairebé 4 hores. Els resultats principals són:

#	**Flags**	**Mitjana (s)**	**Std (s)**	**Min (s)**	**Max (s)**
1	FI,FPL,BPE,BPO,BPH,UWW,UWB	3.474	0.239	3.255	5.606
2	FPL,BPE,BPO,BPH,UWW,UWB	3.484	0.199	3.237	4.644
3	FI,FPL,BPE,BPH,UWW,UWB	3.551	1.147	2.259	8.818
118	NONE	12.986	1.211	10.024	15.756
126	BPO	13.148	1.255	10.154	15.535
127	FPL	13.269	1.221	8.200	16.572
128	FI,BPO,UWW	13.314	1.071	8.202	18.347

Taula 2: Top 3 i Bottom 3 combinacions del TEST_001 (Test A)

Anàlisi dels millors resultats:

- La millor combinació aconsegueix 3.474s de mitjana
- Speedup respecte al seqüencial: $\frac{12.986}{3.474} = 3.74$x
- Les tres millors combinacions són molt similars en temps (3.47-3.55s)
- Totes contenen els tags `FPL`, `BPH`, `UWW` i `UWB`

Anàlisi dels pitjors resultats:

- Les pitjors combinacions són fins i tot més lentes que el seqüencial (NONE)
- La combinació només amb `FPL` dona 13.27s (pitjor que seqüencial!)
- Això indica que alguns tags individuals perjudiquen quan s'usen sols

3.1.5 Variabilitat del Servidor

Un dels descobriments més importants del test va ser l'alta variabilitat del servidor:

Mètrica	Combinació	Valor
Temps mínim absolut	FPL,BPO,UWW	1.908s
Desviació estàndard mínima	FPL,BPE,BPO,BPH,UWW,UWB	0.199s
Desviació estàndard màxima	FI,FPL,BPH,UWW,UWB	2.844s
Desviació relativa màxima	FI,FPL,BPE,BPO,BPH,UWW	42.5%
Range ratio màxim	FPL,BPH,UWW,UWB	9.16x

Taula 3: Estadístiques de variabilitat del servidor (100 execucions)

Conclusions sobre variabilitat:

- **Desviació estàndard**: Entre 0.2s i 2.8s segons la combinació
- **Desviació relativa**: Fins a 42%, significa que 1 de cada 3 execucions pot variar més del 40%
- **Range ratio**: El temps màxim pot ser 9 vegades superior al mínim
- **Implicació**: Només millores superiors al 10% són mesurables de forma fiable

3.1.6 Tests Addicionals (B, C, D, E)

Per confirmar els resultats, es van fer 4 execucions addicionals amb només 1 repetició cadascuna:

Test	Top 3 Combinacions	Temps (s)
B	FPL,BPH,UWW	2.084
	FPL,BPE,BPH,UWW	2.134
	FI,FPL,BPE,BPH,UWW,UWB	2.163
C	FPL,BPH,UWB	1.968
	FPL,BPH,UWW	2.032
	FI,FPL,BPE,BPH,UWW	2.148
D	FPL,BPE,BPH,UWW	2.054
	FI,FPL,BPH,UWW	2.098
	FPL,BPH,UWW,UWB	2.100
E	**FPL,BPH,UWW**	**1.917**
	FPL,BPO,UWB	2.022
	FPL,BPO,UWW	2.033

Taula 4: Millors resultats dels tests individuals B, C, D i E

Observacions:

- Els temps són millors que al test A (1.92-2.16s vs 3.47s)
- Això confirma l'alta variabilitat del servidor
- El millor temps absolut: **1.917s** (Test E)

3.1.7 Freqüència dels Tags

Analitzant els 12 millors resultats dels 5 tests (3 millors $\times$ 4 tests individuals):

Tag	Aparicions	Freqüència
FPL	12 / 12	100%
BPH	10 / 12	83%
UWW	10 / 12	83%
BPE	4 / 12	33%
UWB	4 / 12	33%
FI	3 / 12	25%
BPO	2 / 12	17%

Taula 5: Freqüència d'aparició dels tags en les millors combinacions

Conclusions clares:

- **FPL** apareix en TOTES les millors combinacions (100%)
- **BPH** i **UWW** apareixen en el 83% dels casos
- Aquests tres tags són els candidats clars per a la versió optimitzada
- Els altres tags apareixen menys freqüentment i no semblen essencials

3.1.8 Conclusions del TEST_001

1. **Variabilitat del servidor**: És el factor més limitant, amb variacions de fins a 9x entre execucions
2. **Millor combinació**: Les que inclouen FPL, BPH i UWW
3. **Speedup global**: Aproximadament 3.7x respecte al seqüencial (amb variabilitat)
4. **Confiança estadística**: El test A (100 repeticions) dona resultats fiables malgrat la variabilitat
5. **Necessitat de validació**: Cal confirmar amb el TEST_002 que els speedups individuals d'aquests tags són realment positius

3.2 TEST_002: Speedup Individual

3.2.1 Objectiu

Mesurar el speedup de cada paral·lelització de forma aïllada per confirmar quines són realment beneficioses i coherents amb els resultats del TEST_001.

3.2.2 Metodologia

- Es modifica `training.c` per inserir `omp_get_wtime()` abans i després de cada bucle paral·lelitzat
- Es fa una còpia de `training.c` a `TESTS/TEST_002/training.c`
- S'executa el codi amb `ALL` (totes les paral·lelitzacions actives)
- S'executa el codi seqüencial (sense cap paral·lelització)
- Es calcula el speedup: $S = \frac{T_{\text{seq}}}{T_{\text{par}}}$

Exemple d'instrumentació:

```
void feed_input(int i) {
    double start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int j = 0; j < num_neurons[0]; j++)
        lay[0].actv[j] = input[i][j];
    printf("FI: %18.16f\n", omp_get_wtime() - start_time);
}
```

Listing 13: Example Code

3.2.3 Execució

```
python3 run_test.py 002 [SUBFOLDER] [MODE]
```

Listing 14: Example Code

El script:

- Compila amb `ALL` i executa
- Compila sense flags i executa
- Llegeix els fitxers `.out` i extreu els temps
- Calcula speedup i eficiència per cada tag

3.2.4 Resultats del Test A (primera execució)

#	**Tag**	$T_{\mathbf{par}}$ **(s)**	$T_{\mathbf{seq}}$ **(s)**	**Speedup**	**Fracció seq.**
1	BPH	0.360	3.158	**8.77**	30.5%
2	UWW	0.335	2.639	**7.87**	25.5%
3	FPL	1.124	4.483	**3.99**	43.3%
4	FI	0.091	0.016	0.18	0.2%
5	UWB	0.185	0.025	0.13	0.2%
6	BPO	0.903	0.034	0.04	0.3%
7	BPE	0.089	0.003	0.04	0.03%

Taula 6: Speedup individual per tag (Test A del TEST_002)

Anàlisi detallada:

- **BPH** (Backpropagation Hidden):
  - Millor speedup: 8.77x
  - Representa el 30.5% del temps seqüencial total
  - Temps seqüencial: 3.158s (molt treball)
  - Temps paral·lel: 0.360s
  - Eficiència: $\frac{8.77}{12} = 0.73$ (73%)

- **UWW** (Update Weights):
  - Segon millor speedup: 7.87x
  - Representa el 25.5% del temps seqüencial
  - Molt treball per actualitzar tots els pesos
  - Eficiència: 0.66 (66%)
- **FPL** (Forward Propagation):
  - Tercer speedup: 3.99x
  - Representa el 43.3% del temps seqüencial (LA MÉS GRAN!)
  - Tot i tenir menys speedup, és la secció més important
  - Eficiència: 0.33 (33%)
- **Tags amb speedup < 1**:
  - FI (0.18): Temps paral·lel (0.091s) »> Temps seqüencial (0.016s)
  - UWB (0.13): Overhead de threads supera el benefici
  - BPO (0.04): 25 vegades més lent amb paral·lelització!
  - BPE (0.04): Només representa 0.03% del temps total

3.2.5 Resultats Mitjans (Tests A-E)

Després de 5 execucions, la mitjana és:

#	Tag	T_{par} (s)	T_{seq} (s)	Speedup	Eficiència
1	BPH	0.331	3.156	**9.55**	0.80
2	UWW	0.347	2.623	**7.55**	0.63
3	FPL	1.124	4.480	**3.99**	0.33
4	FI	0.102	0.016	0.16	-
5	UWB	0.195	0.025	0.13	-
6	BPO	0.959	0.034	0.04	-
7	BPE	0.092	0.003	0.03	-

Taula 7: Speedup i eficiència mitjans (5 execucions)

Observacions:

- Els resultats són molt consistents entre execucions
- La mitjana millora lleugerament els resultats individuals (BPH: 8.77 → 9.55)
- La separació clara entre tags bons (>1) i dolents (<1) es manté

3.2.6 Coherència amb TEST_001

Tag	Freqüència TEST_001	Speedup TEST_002
FPL	100%	3.99
BPH	83%	9.55
UWW	83%	7.55
BPE	33%	0.03
UWB	33%	0.13
FI	25%	0.16
BPO	17%	0.04

Taula 8: Coherència entre freqüència d'aparició i speedup individual

Conclusió: Hi ha una correlació perfecta! Els tags que apareixen més freqüentment al TEST_001 són exactament els que tenen speedup > 1 al TEST_002.

3.2.7 Per Què Alguns Tags Apareixen al Top del TEST_001?

Tot i tenir speedup < 1, tags com `BPE` apareixen en algunes combinacions del top 10. Per què?

- **Impacte mínim**: Representen menys de l'1% del temps total
- **Variabilitat del servidor**: Les variacions de 0.2-2.8s amaguen l'empitjorament de 0.09s
- **Exemple**: Si una combinació guanya 7s amb BPH+UWW+FPL però perd 0.09s amb BPE, el resultat net segueix sent molt bo

3.2.8 Fracció Paral·lelitzable Total

Sumant les fraccions seqüencials dels tres millors tags:

$$f = 0.305 + 0.254 + 0.433 = 0.992 = 99.2\% \tag{5}$$

Això significa que gairebé tot el temps d'execució és paral·lelitzable, la qual cosa explica els bons speedups obtinguts.

3.2.9 Conclusions del TEST_002

1. **Validació del TEST_001**: Els resultats són coherents amb les freqüències observades
2. **Tags descartats**: FI, UWB, BPO i BPE fan el programa més lent
3. **Tags seleccionats**: BPH, UWW i FPL constitueixen la versió `OPT`
4. **Millor eficiència**: BPH amb 80% d'aprofitament dels threads
5. **Fracció paral·lelitzable**: 99.2% del temps és paral·lelitzable

6. **Speedup teòric**: Segons Amdahl, amb $f = 0.992$ i $p = 12$: $S_{\max} = \frac{1}{0.008 + \frac{0.992}{12}} = 11.76$x

A partir d'ara, `OPT` = `FPL` + `BPH` + `UWW`

3.3 TEST_003: Escalabilitat i Configuracions

3.3.1 Objectiu

Avaluar el rendiment de la versió `OPT` sota diferents configuracions:

- Diferents servidors
- Diferents particions (`nodo.q` i `new-nodo.q`)
- Nombre de threads (1, 2, 4, 6, 8, 10, 12)
- Nombre d'èpoques (1, 10, 100, 1000)
- Nombre de neurones a la capa oculta (135, 250)

3.3.2 Metodologia

Per aquest test mantenim la paral·lelització òptima trobada (`OPT` = `FPL` + `BPH` + `UWW`) i variem les altres configuracions.

Considerem com a temps seqüencial el temps amb optimització `OPT` però amb només un thread, ja que és més fàcil des del punt de vista programàtic i també és equivalent a no tenir paral·lelització.

Automatització:

- El fitxer `scheduler.sub` es modifica per cada execució
- Els fitxers de configuració es copien a `TESTS/TEST_003/configuration`
- Aquesta carpeta es pot eliminar després, ja que les configuracions queden emmagatzemades als metadades de cada fitxer de sortida

Script d'execució:

```
python3 run_test.py 003 [SUBFOLDER] [MODE] [VARIATE]
```

Listing 15: Example Code

On els paràmetres són:

- `SUBFOLDER`: subcarpeta dins de `OUT/` per als fitxers de sortida (buit o `X` crea la següent lletra)
- `MODE`: `e` per executar, `a` per analitzar, o ambdós
- `VARIATE`: flags que controlen les variacions:
  - `P`: Particions
  - `T`: Threads
  - `E`: Èpoques
  - `N`: Neurones

3.3.3 Execució del Test A

Per la primera prova executem totes les variacions al servidor Wilma:

```
python3 run_test.py 003 A ea PTEN
```

Listing 16: Example Code

Això varia tots els paràmetres possibles i guarda les sortides a `TESTS/TEST_003/OUT/A`.

3.3.4 Resultats: Servidor Wilma - Partició nodo.q

Threads	1 època	10 èpoques	100 èpoques	1000 èpoques
1	1.26s	10.44s	102.25s	1020.58s
2	0.65s	5.24s	50.56s	508.68s
4	0.36s	2.84s	27.58s	273.89s
6	0.28s	2.16s	21.03s	209.26s
8	0.25s	1.86s	18.08s	182.33s
10	0.26s	1.92s	18.13s	179.97s
12	0.27s	2.12s	20.19s	199.30s

Taula 9: Temps d'execució (135 neurones, nodo.q)

Configuració: 135 neurones Anàlisi del speedup:

- Amb 8 threads s'aconsegueix el millor speedup: **5.66x** (1000 èpoques)
- El speedup millora amb més èpoques (Llei de Gustafson)
- Amb 10 i 12 threads el rendiment empitjora lleugerament (overhead)

Anàlisi de l'eficiència:

- Amb 2 threads: eficiència $> 99\%$ (gairebé ideal!)
- Amb 4 threads: eficiència $\approx 93\%$
- Amb 8 threads: eficiència $\approx 70\%$
- Amb 12 threads: eficiència $\approx 43\%$

Threads	1 època	10 èpoques	100 èpoques	1000 èpoques
1	2.28s	19.08s	186.78s	1865.72s
2	1.19s	9.84s	95.26s	960.22s
4	0.62s	4.88s	47.30s	473.81s
6	0.44s	3.49s	33.98s	337.01s
8	0.37s	2.86s	27.81s	276.76s
10	0.33s	2.55s	25.04s	248.49s
12	0.31s	2.37s	23.27s	240.32s

Taula 10: Temps d'execució (250 neurones, nodo.q)

Configuració: 250 neurones Millores respecte 135 neurones:

- Millor speedup: **8.05x** amb 12 threads i 10 èpoques
- L'eficiència es manté alta: 84% amb 8 threads
- Augmentar les neurones incrementa la part paral·lelitzable → millor speedup
- Els resultats són més consistents (menys fluctuacions)

3.3.5 Resultats: Servidor Wilma - Partició new-nodo.q

Threads	1 època	10 èpoques	100 èpoques	1000 èpoques
1	0.38s	3.09s	30.18s	302.59s
2	0.22s	1.73s	16.54s	163.80s
4	0.17s	1.37s	12.16s	119.46s
6	0.15s	1.02s	9.79s	98.08s
8	0.12s	0.91s	9.91s	88.62s
10	0.14s	1.19s	9.16s	89.98s
12	0.15s	1.06s	9.32s	90.67s

Taula 11: Temps d'execució (135 neurones, new-nodo.q)

Configuració: 135 neurones Observacions clau:

- Temps aproximadament **3x més ràpids** que amb `nodo.q`
- Millor temps absolut: 0.12s (8 threads, 1 època)
- Speedup màxim més modest: **3.41x** amb 8 threads
- L'eficiència és més baixa (43% amb 8 threads)

Threads	**1 època**	**10 èpoques**	**100 èpoques**	**1000 èpoques**
1	0.69s	5.98s	56.74s	569.24s
2	0.39s	3.04s	29.61s	294.12s
4	0.25s	2.15s	18.87s	188.79s
6	0.19s	1.81s	15.04s	150.01s
8	0.20s	1.48s	13.37s	128.72s
10	0.20s	1.70s	12.75s	123.96s
12	0.20s	1.17s	11.71s	116.76s

Taula 12: Temps d'execució (250 neurones, new-nodo.q)

Configuració: 250 neurones Millores amb 250 neurones:

- Millor speedup: **5.09x** amb 12 threads i 10 èpoques
- Eficiència millor que amb 135 neurones (55% amb 8 threads)
- Segueix sent inferior a `nodo.q` en termes de speedup relatiu

3.3.6 Validació de Resultats

El nombre d'encerts (hits) es manté constant per totes les configuracions amb el mateix nombre d'èpoques i neurones, confirmant que la paral·lelització no afecta la correcció:

Neurones	1 època	10 èpoques	100 èpoques	1000 èpoques
135	787	885	903	911
250	810	898	919	918

Taula 13: Nombre d'encerts (consistents en totes les execucions)

Implicació: Totes les execucions han aconseguit els mateixos resultats, independentment del nombre de threads utilitzats. Això valida que la implementació paral·lela és correcta.

3.3.7 Anàlisi Comparativa

Impacte de les èpoques

- **1 època**: Speedup baix (1.9-3.5x) per l'overhead de creació de threads
- **10 èpoques**: Speedup millora significativament (5.4-8.1x)
- **100-1000 èpoques**: Speedup s'estabilitza (5.0-8.0x)
- **Conclusió**: Més èpoques → major fracció paral·lelitzable (Llei de Gustafson)

Partició	135 neurones	250 neurones	Millora
nodo.q	5.66x	8.05x	+42%
new-nodo.q	3.41x	5.09x	+49%

Taula 14: Speedup màxim segons nombre de neurones

Impacte de les neurones **Explicació**: Augmentar les neurones incrementa el treball als bucles paral·lelitzats (especialment BPH i FPL), disminuint la fracció seqüencial.

Mètrica	**nodo.q**	**new-nodo.q**
Temps absolut (250n, 1000è)	1865.72s	569.24s
Reducció de temps	-	3.3x més ràpid
Speedup màxim (250n)	8.05x	5.09x
Eficiència màxima (250n)	84% (8t)	55% (8t)

Taula 15: Comparació entre particions (250 neurones)

Comparació de particions Interpretació:

- `new-nodo.q` té processadors més ràpids (temps absoluts millors)
- Però la part seqüencial és més ràpida també → menor marge per paral·lelitzar
- Segons la Llei d'Amdahl: si la part seqüencial és molt ràpida, el speedup màxim es redueix

3.3.8 Comportament de l'Eficiència

L'eficiència disminueix amb més threads seguint la Llei d'Amdahl:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{6}$$

On f és la fracció paral·lelitzable i p el nombre de threads.

Exemple amb 250 neurones, 1000 èpoques, nodo.q:

- 2 threads: eficiència = 97% (gairebé lineal)
- 4 threads: eficiència = 98% (encara millor!)
- 8 threads: eficiència = 84%
- 12 threads: eficiència = 67%

El fet que l'eficiència amb 4 threads sigui superior a la de 2 threads suggereix efectes de cache i millor distribució de càrrega.

3.3.9 Nombre Òptim de Threads

Configuració	Threads òptims	Speedup	Eficiència
135n, nodo.q	8-10	5.66x	70%
250n, nodo.q	10-12	7.76x	65-67%
135n, new-nodo.q	8	3.41x	43%
250n, new-nodo.q	12	5.09x	42%

Taula 16: Configuració òptima per cada escenari

Recomanació: Per `nodo.q`, utilitzar 8-10 threads ofereix el millor equilibri entre speedup i eficiència.

3.3.10 Conclusions del TEST_003

1. **Llei de Gustafson confirmada**: Augmentar les èpoques millora significativament l'eficiència de la paral·lelització (de 3.5x amb 1 època a 8.0x amb 1000 èpoques).

2. **Impacte de les neurones**: Més neurones → major speedup. Passar de 135 a 250 neurones millora el speedup entre un 42-49%.

3. **Diferències entre particions**:

   - `new-nodo.q` és 3x més ràpid en temps absolut
   - Però `nodo.q` aconsegueix millors speedups relatius (8.05x vs 5.09x)
   - Causa: processadors més ràpids redueixen el marge de millora per paral·lelització

4. **Llei d'Amdahl verificada**: L'eficiència disminueix amb més threads de manera previsible. L'overhead de sincronització es fa significatiu amb 10-12 threads.

5. **Eficiències excel·lents**: Amb configuracions adequades (250 neurones, 1000 èpoques, 4 threads) s'aconsegueixen eficiències de 98%, gairebé ideal.

6. **Correcció garantida**: El nombre d'encerts és idèntic en totes les execucions, independent del nombre de threads. La paral·lelització no introdueix errors.

7. **Escalabilitat demostrada**: El codi escala bé fins a 8-10 threads. Més enllà, l'overhead supera els beneficis.

8. **Configuració recomanada**: Per màxim rendiment amb eficiència acceptable:

   - Partition: `nodo.q` (millor speedup relatiu)
   - Threads: 8-10
   - Neurones: 250+ (més és millor)
   - Èpoques: 100+ (més és millor per paral·lelització)

El TEST_003 confirma que la versió `OPT` és robusta, escalable i correcta sota una àmplia gamma de configuracions, validant les decisions preses als tests anteriors.

4 Discussió

4.1 Interpretació dels Resultats

4.1.1 Overhead de Paral·lelització

Els resultats del TEST_002 revelen clarament que no totes les paral·lelitzacions són beneficioses. Les seccions `FI`, `UWB`, `BPO` i `BPE` presenten speedups inferiors a 1, amb valors tan baixos com 0.03.

Això s'explica pel fet que aquestes seccions:

- Representen menys de l'1% del temps total cadascuna
- Tenen bucles molt curts (poques iteracions)
- L'overhead de creació i sincronització de threads ($\sim$0.1 ms) supera el temps de còmput

La relació entre temps de còmput i overhead es pot expressar com:

$$\text{Speedup} = \frac{T_{\text{còmput}}}{T_{\text{overhead}} + \frac{T_{\text{còmput}}}{p}} \tag{7}$$

Quan $T_{\text{còmput}} \ll T_{\text{overhead}}$, el speedup serà sempre < 1.

4.1.2 Llei d'Amdahl en Acció

Els resultats del TEST_003 mostren clarament els efectes de la Llei d'Amdahl. Tot i tenir una fracció paral·lelitzable del 99.2%, el speedup amb 12 threads és de només 4.97 (eficiència 0.414).

Calculant la fracció seqüencial real:

$$\alpha = \frac{1 - \frac{S}{p}}{1 - \frac{1}{p}} = \frac{1 - \frac{4.97}{12}}{1 - \frac{1}{12}} \approx 0.454 \tag{8}$$

Això indica que, en la pràctica, el 45.4% del temps no es beneficia de la paral·lelització, molt més que l'0.8% teòric. Les causes inclouen:

- Overhead de creació i sincronització de threads
- Regions crítiques (`#pragma omp critical`)
- Desbalanceig de càrrega entre threads
- Contenciósns en memòria cau

4.1.3 Llei de Gustafson i Escalabilitat

Els millors resultats amb més neurones (speedup 7.77 vs 4.97) i més epochs (5.53 vs 4.97) confirmen la Llei de Gustafson: quan la mida del problema creix, la fracció paral·lelitzable augmenta i el speedup millora.

Configuració	**Speedup (12 threads)**	**Eficiència**
135 neurons, 10 epochs	4.97	0.414
250 neurons, 10 epochs	7.77	0.648
135 neurons, 50 epochs	5.53	0.461

Figura 1: Impacte de la mida del problema en l'escalabilitat

4.2 Limitacions i Reptes

4.2.1 Variabilitat del Servidor

La variabilitat observada (desviacions de fins a 2.8 segons, range ratios de 9.16) limita la capacitat de mesurar optimitzacions petites. Això implica que:

- Només millores $> 10\%$ són estadísticament significatives
- Es requereixen moltes repeticions per obtenir resultats fiables
- Comparacions directes entre execucions individuals no són vàlides

4.2.2 Contencions de Memòria

La secció `BPH` utilitza `#pragma omp critical` per evitar condicions de cursa:

```
if (i > 1) {
    #pragma omp critical
    lay[i - 1].dactv[k] += ...;
}
```

Listing 17: Example Code

Aquesta regió crítica serialitza parcialment el codi, limitant el speedup teòric. Tot i això, com que només s'executa per $i > 1$, l'impacte és moderat.

4.2.3 Saturació amb Molts Threads

A `new-nodo.q`, s'observa saturació a partir de 48 threads amb 135 neurones. Això es deu a:

- Insuficient paral·lelisme disponible (bucles massa curts)
- Contenciósns en memòria compartida
- Overhead de gestió de molts threads
- Limitacions de l'arquitectura (nombre de nuclis físics)

5 Conclusions

Aquest estudi ha demostrat l'eficàcia de la paral·lelització amb OpenMP per a l'entrenament de xarxes neuronals, amb les següents conclusions principals:

1. **Identificació de seccions crítiques**: De les set paral·lelitzacions candidates, només tres (`FPL`, `BPH`, `UWW`) aporten millores significatives, amb speedups individuals de 3.99x, 9.55x i 7.55x respectivament.

2. **Speedup global**: La versió optimitzada (`OPT`) aconsegueix un speedup de 4.97x amb 12 threads i 135 neurones, arribant a 7.77x amb 250 neurones.

3. **Eficiència decreixent**: L'eficiència passa de 0.845 amb 2 threads a 0.414 amb 12 threads, confirmant les prediccions de la Llei d'Amdahl.

4. **Escalabilitat positiva**: Augmentar la mida del problema (més neurones o epochs) millora tant el speedup com l'eficiència, validant la Llei de Gustafson.

5. **Limitacions pràctiques**: L'overhead de paral·lelització i la variabilitat del servidor són factors limitants importants que cal considerar.

Aquest projecte ha demostrat que la paral·lelització amb OpenMP és una eina efectiva per accelerar l'entrenament de xarxes neuronals, aconseguint speedups de fins a 7.77x en configuracions realistes. Tanmateix, els resultats també posen de manifest les limitacions intrínseques de la paral·lelització en memòria compartida i la importància d'una anàlisi rigorosa per identificar les seccions que realment es beneficien de la paral·lelització.

Els coneixements adquirits són aplicables no només a xarxes neuronals, sinó a qualsevol aplicació científica o d'enginyeria que requereixi computació intensiva. La metodologia desenvolupada, amb la seva combinació d'automatització, mesura precisa i anàlisi estadística, constitueix un marc robust per a futurs estudis d'optimització.

6 Annex: Resum de l'output dels TESTs

#	avg	std	min	max	n	rel_std	range_ratio	flags
001	3.473646	0.238805	3.255061	5.605700	100	0.068748	1.722149	FI, FPL, BPE, BPO, BPH, UWW, UWB
002	3.484115	0.198727	3.236850	4.643671	100	0.057038	1.434627	FPL, BPE, BPO, BPH, UWW, UWB
003	3.550909	1.146911	2.258562	8.818422	100	0.322991	3.904441	FI, FPL, BPE, BPH, UWW, UWB
...	...	...	...	...	...	...	...	...
117	12.979604	1.141285	10.125176	15.171276	100	0.087929	1.498372	BPH
118	12.985589	1.211251	10.023879	15.756089	99	0.093277	1.571855	NONE
119	13.020803	1.287519	10.779306	21.959309	100	0.098882	2.037173	FI, FPL, UWB
...	...	...	...	...	...	...	...	...
126	13.147614	1.254910	10.153961	15.535125	100	0.095448	1.529957	BPO
127	13.269134	1.221293	8.199807	16.571745	100	0.092040	2.020992	FPL
128	13.314294	1.070659	8.201717	18.346678	100	0.080414	2.236931	FI, BPO, UWW

Taula 17: TEST 001 (A)

#	avg	std	min	max	n	rel_std	range_ratio	flags
039	9.410032	1.511699	1.908416	13.167567	100	0.160648	6.899736	FPL, BPO, UWW
002	3.484115	0.198727	3.236850	4.643671	100	0.057038	1.434627	FPL, BPE, BPO, BPH, UWW, UWB
020	8.253980	2.844047	2.177416	15.607096	100	0.344567	7.167714	FI, FPL, BPH, UWW, UWB
006	3.683354	1.566816	2.330171	11.681657	100	0.425377	5.013219	FI, FPL, BPE, BPO, BPH, UWW
067	11.063307	2.784355	2.189258	20.044112	100	0.251675	9.155665	FPL, BPH, UWW, UWB

Taula 18: TEST 001 (A) estadístiques

#	avg	std	min	max	n	rel_std	range_ratio	flags
001	2.084486	0.000000	2.084486	2.084486	1	0.000000	1.000000	FPL, BPH, UWW
002	2.134077	0.000000	2.134077	2.134077	1	0.000000	1.000000	FPL, BPE, BPH, UWW
003	2.162706	0.000000	2.162706	2.162706	1	0.000000	1.000000	FI, FPL, BPE, BPH, UWW, UWB
...	...	...	...	...	...	...	...	...
001	1.968112	0.000000	1.968112	1.968112	1	0.000000	1.000000	FPL, BPH, UWB
002	2.032400	0.000000	2.032400	2.032400	1	0.000000	1.000000	FPL, BPH, UWW
003	2.147710	0.000000	2.147710	2.147710	1	0.000000	1.000000	FI, FPL, BPE, BPH, UWW
...	...	...	...	...	...	...	...	...
001	2.054270	0.000000	2.054270	2.054270	1	0.000000	1.000000	FPL, BPE, BPH, UWW
002	2.097778	0.000000	2.097778	2.097778	1	0.000000	1.000000	FI, FPL, BPH, UWW
003	2.100028	0.000000	2.100028	2.100028	1	0.000000	1.000000	FPL, BPH, UWW, UWB
...	...	...	...	...	...	...	...	...
001	1.916558	0.000000	1.916558	1.916558	1	0.000000	1.000000	FPL, BPH, UWW
002	2.022203	0.000000	2.022203	2.022203	1	0.000000	1.000000	FPL, BPO, UWB
003	2.032688	0.000000	2.032688	2.032688	1	0.000000	1.000000	FPL, BPO, UWW

Taula 19: TEST 001 (B, C, D i E)

#	time_par	fraction_par	time_seq	fraction_seq	speedup	tag
1	0.359971	0.116600	3.158224	0.304900	8.7736	BPH
2	0.335285	0.108600	2.638832	0.254800	7.8704	UWW
3	1.123978	0.364000	4.482735	0.432800	3.9883	FPL
4	0.091257	0.029600	0.016294	0.001600	0.1786	FI
5	0.185019	0.059900	0.024912	0.002400	0.1346	UWB
6	0.902893	0.292400	0.033675	0.003300	0.0373	BPO
7	0.089107	0.028900	0.003167	0.000300	0.0355	BPE

Taula 20: TEST 002 (A)

#	time_par	fraction_par	time_seq	fraction_seq	speedup	tag
001	0.331066	0.1052	3.156015	0.3053	9.5527	BPH
002	0.347434	0.1103	2.622788	0.2538	7.5549	UWW
003	1.124091	0.3569	4.479600	0.4334	3.9865	FPL
004	0.101662	0.0323	0.016016	0.0016	0.1584	FI
005	0.195332	0.0620	0.024630	0.0024	0.1264	UWB
006	0.958994	0.3042	0.033569	0.0033	0.0351	BPO
007	0.092299	0.0293	0.002868	0.0003	0.0312	BPE

Taula 21: TEST 002 (mitjana A, B, C, D i E)

Threads	1	10	100	1000
1	1.262934	10.444521	102.246777	1020.584532
2	0.650277	5.240709	50.557287	508.681681
4	0.364041	2.836148	27.579601	273.890295
6	0.282699	2.162346	21.033529	209.262108
8	0.247138	1.857143	18.075211	182.326626
10	0.256245	1.918224	18.129956	179.969913
12	0.270800	2.121950	20.193809	199.299296

Taula 22: TEST 003 Runtime (threads x epochs) - 135 Neurons - nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	1.9421	1.9930	2.0224	2.0063
4	3.4692	3.6826	3.7073	3.7263
6	4.4674	4.8302	4.8611	4.8771
8	5.1102	5.6240	5.6567	5.5976
10	4.9286	5.4449	5.6397	5.6709
12	4.6637	4.9221	5.0633	5.1209

Taula 23: TEST 003 Speedup - 135 Neurons - nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	0.9711	0.9965	1.0112	1.0032
4	0.8673	0.9207	0.9268	0.9316
6	0.7446	0.8050	0.8102	0.8128
8	0.6388	0.7030	0.7071	0.6997
10	0.4929	0.5445	0.5640	0.5671
12	0.3886	0.4102	0.4219	0.4267

Taula 24: TEST 003 Efficiency (speedup / threads) - 135 Neurons - nodo.q

Threads	1	10	100	1000
1	2.283024	19.079364	186.779419	1865.718895
2	1.193518	9.841327	95.256268	960.223371
4	0.615179	4.877443	47.295550	473.813401
6	0.443950	3.494337	33.984099	337.006243
8	0.370750	2.861059	27.813612	276.763356
10	0.331393	2.546710	25.039208	248.487582
12	0.312266	2.369382	23.266886	240.322038

Taula 25: TEST 003 Runtime (threads x epochs) - 250 Neurons - nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	1.9129	1.9387	1.9608	1.9430
4	3.7112	3.9118	3.9492	3.9377
6	5.1425	5.4601	5.4961	5.5362
8	6.1579	6.6686	6.7154	6.7412
10	6.8892	7.4918	7.4595	7.5083
12	7.3112	8.0525	8.0277	7.7634

Taula 26: TEST 003 Speedup - 250 Neurons - nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	0.9564	0.9693	0.9804	0.9715
4	0.9278	0.9779	0.9873	0.9844
6	0.8571	0.9100	0.9160	0.9227
8	0.7697	0.8336	0.8394	0.8427
10	0.6889	0.7492	0.7459	0.7508
12	0.6093	0.6710	0.6690	0.6470

Taula 27: TEST 003 Efficiency (speedup / threads) - 250 Neurons - nodo.q

Neurons	1	10	100	1000
135	787.0000	885.0000	903.0000	911.0000
250	810.0000	898.0000	919.0000	918.0000

Taula 28: TEST 003 Hits Table - nodo.q

Threads	1	10	100	1000
1	0.381573	3.089303	30.176114	302.587106
2	0.219941	1.726364	16.541138	163.799335
4	0.172096	1.367126	12.155316	119.457508
6	0.149440	1.017981	9.787883	98.081797
8	0.122050	0.912874	9.906239	88.623003
10	0.144938	1.189781	9.158091	89.982792
12	0.154116	1.057747	9.322235	90.668037

Taula 29: TEST 003 Runtime (threads x epochs) - 135 Neurons - new-nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	1.7349	1.7895	1.8243	1.8473
4	2.2172	2.2597	2.4825	2.5330
6	2.5534	3.0347	3.0830	3.0850
8	3.1264	3.3842	3.0462	3.4143
10	2.6327	2.5965	3.2950	3.3627
12	2.4759	2.9206	3.2370	3.3373

Taula 30: TEST 003 Speedup - 135 Neurons - new-nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	0.8674	0.8947	0.9122	0.9237
4	0.5543	0.5649	0.6206	0.6333
6	0.4256	0.5058	0.5138	0.5142
8	0.3908	0.4230	0.3808	0.4268
10	0.2633	0.2597	0.3295	0.3363
12	0.2063	0.2434	0.2698	0.2781

Taula 31: TEST 003 Efficiency (speedup / threads) - 135 Neurons - new-nodo.q

Threads	1	10	100	1000
1	0.694904	5.977322	56.742845	569.237095
2	0.390218	3.035056	29.613842	294.115588
4	0.247445	2.150360	18.872645	188.793684
6	0.193692	1.811668	15.043629	150.006977
8	0.201809	1.477204	13.373775	128.719548
10	0.201103	1.695738	12.752251	123.963191
12	0.197376	1.174499	11.705240	116.756898

Taula 32: TEST 003 Runtime (threads x epochs) - 250 Neurons - new-nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	1.7808	1.9694	1.9161	1.9354
4	2.8083	2.7797	3.0066	3.0151
6	3.5877	3.2993	3.7719	3.7947
8	3.4434	4.0464	4.2428	4.4223
10	3.4555	3.5249	4.4496	4.5920
12	3.5207	5.0893	4.8476	4.8754

Taula 33: TEST 003 Speedup - 250 Neurons - new-nodo.q

Threads	1	10	100	1000
1	1.0000	1.0000	1.0000	1.0000
2	0.8904	0.9847	0.9580	0.9677
4	0.7021	0.6949	0.7517	0.7538
6	0.5979	0.5499	0.6286	0.6325
8	0.4304	0.5058	0.5304	0.5528
10	0.3455	0.3525	0.4450	0.4592
12	0.2934	0.4241	0.4040	0.4063

Taula 34: TEST 003 Efficiency (speedup / threads) - 250 Neurons - new-nodo.q

Neurons	1	10	100	1000
135	787.0000	885.0000	903.0000	911.0000
250	810.0000	898.0000	919.0000	918.0000

Taula 35: TEST 003 Hits Table - new-nodo.q

7 Glossari

Backpropagation Algoritme per calcular el gradient de la funció de pèrdua respecte als pesos de la xarxa neuronal, propagant l'error des de la sortida cap a l'entrada.

Bias (Biaix) Paràmetre addicional en cada neurona que permet desplaçar la funció d'activació, millorant la capacitat d'aprenentatge del model.

Critical Section (Regió Crítica) Fragment de codi que només pot ser executat per un thread alhora per evitar condicions de cursa.

Eficiència Mesura de l'aprofitament dels recursos computacionals, definida com el speedup dividit pel nombre de processadors.

Epoch Una iteració completa sobre tot el conjunt d'entrenament durant l'entrenament d'una xarxa neuronal.

Forward Propagation Procés de calcular la sortida de la xarxa neuronal propagant les entrades cap endavant a través de les capes.

Gradient Descent Mètode d'optimització que ajusta els pesos de la xarxa en la direcció oposada al gradient per minimitzar la funció de pèrdua.

Hidden Layer (Capa Oculta) Capa intermèdia d'una xarxa neuronal entre l'entrada i la sortida, que extreu característiques complexes.

Llei d'Amdahl Llei que prediu el límit del speedup en funció de la fracció paral·lelitzable del codi.

Llei de Gustafson Llei que suggereix que l'escalabilitat millora quan la mida del problema creix amb els recursos disponibles.

OpenMP API per programació paral·lela en memòria compartida mitjançant directives de compilador.

Overhead Temps addicional necessari per gestionar la paral·lelització (creació de threads, sincronització, etc.).

Race Condition (Condició de Cursa) Error que ocorre quan múltiples threads accedeixen simultàniament a una variable compartida sense sincronització adequada.

RELU (Rectified Linear Unit) Funció d'activació definida com $f(x) = \max(0, x)$, àmpliament utilitzada en xarxes neuronals profundes.

Sigmoid Funció d'activació definida com $f(x) = \frac{1}{1+e^{-x}}$, utilitzada típicament en la capa de sortida per a classificació binària.

SLURM Sistema de gestió de recursos i planificació de tasques per clusters de computació d'altes prestacions.

Speedup Ràtio entre el temps d'execució seqüencial i el temps d'execució paral·lel, que mesura la millora de rendiment.

Thread Unitat bàsica d'execució que pot córrer en paral·lel amb altres threads dins d'un mateix procés.

Weight (Pes) Paràmetre que determina la força de la connexió entre dues neurones en una xarxa neuronal.